BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images may include (but are not limited to):

- BLACK BORDERS
- TEXT CUT OFF AT TOP, BOTTOM OR SIDES
- FADED TEXT
- ILLEGIBLE TEXT
- SKEWED/SLANTED IMAGES
- COLORED PHOTOS
- BLACK OR VERY BLACK AND WHITE DARK PHOTOS
- GRAY SCALE DOCUMENTS

IMAGES ARE BEST AVAILABLE COPY.

As rescanning documents *will not* correct images, please do not report the images to the Image Problem Mailbox.

REFERENCE

Troux 4

Troux Query Language Reference

Version 4.0.1
January 31, 2004

Troux 4
Troux Query Language Reference
Product Version 4.0.1
Publication Date: January 29, 2004

Table of Contents

How to Use This Document 1

Chapter 1 Understanding the Basics 3
- TQL Format 4
- Conjunctions 5
- Using Property and Category Names 5
- Using Values 5
- Using Operators 6

Chapter 2 Working with Components 7
- Working With Component Core Attributes 8
  - component.id 8
  - component.name 8
  - component.type 8
  - component.exactType 9
  - component.property 9
  - component.hasEvent 10
- Working With Component Hierarchies 10
  - component.topLevel 10
  - component.hasAncestor 11
  - component.hasDescendant 11
  - component.hasParent 11
- Working With Component Sets 11
  - define.component.set 11
  - component.set 11
- Working With Relationship in Component Queries 12
  - component.hasRelationship 12
  - component.relatedTo 12
  - component.dependsOn 12
  - component.requiredFor 12
- Working With Component States 13
  - component.abstractType 13
  - component.disabledType 13
  - component.hasMissingRequiredProperty 13
  - component.hasInvalidProperty 13

Chapter 3 Working With Relationships 15
Working With Relationship Core Attributes 16
relationship.id 16
relationship.type 16
relationship.exactType 16
relationship.property 17
relationship.hasEvent 17
Working With Relationships' Components 18
relationship.hasComponent 18
relationship.traceRelationshipsFrom 18
Working With Relationship Sets 18
define.relationship.set 18
relationship.set 18
Working With Relationship States 19
relationship.disabledType 19
relationship.hasMissingRequiredProperty 19
relationship.hasInvalidProperty 19

Chapter 4 Using Parameters and Parameter Selectors 21
define.parameter 22
parameter 22
Working With Parameter Selectors 22
selectOperator 22
selectDate 23
selectExistingComponentPropertyValue 24
selectPossibleComponentPropertyValue 24
selectExistingRelationshipPropertyValue 24
selectPossibleRelationshipPropertyValue 25
selectComponentType 25
selectRelationshipType 25
enumeration 26
Working with Data Type-Dependent Selectors 26
selectComponentPropertyByType 26
selectRelationshipPropertyByType 27

Appendix A Example Views 29
View A. Exploring a Database Server's Related Components 29
TQL Query 29
Query Explanation 29
Resulting View 30
View B. Exploring an Application's Subcomponents 31
TQL Query 31
Query Explanation 32
Resulting View 32
View C. Exploring Components Used by Business Unit 1 33
TQL Query 33
Query Explanation 33
Resulting View 34

How to Use This Document

This document describes the use of the Troux Query Language (TQL) that is used in Troux 4 to view data and component details.

Troux recommends that users who are new to Troux 4 review this entire document.

For experienced users of the Troux Blueprinting System, this document will serve as a reference for using TQL.

The document is divided into the following chapters:

- *Understanding the Basics* - provides a short overview of the basic parts of a TQL query.
- *Working with Components* - covers the TQL clauses for working with component attributes.
- *Working With Relationships* - covers the TQL clauses for working with relationship attributes.
- *Working With Parameters* - This chapter covers TQL statements used to create and return parameters associated with components and relationships.
- *Example Views* - This section provides three examples of views written in TQL.

Throughout this document, names of Troux Technologies objects and entities are listed in *italics*. Named items on the user interface such as windows, information boxes, and menus are listed as *Italicized and Capitalized Text*.

Tip: Useful information contained in this document that is helpful to users of all levels is included in this "Tip:" format. The most common use of this format is for timesaving shortcuts and best practices.

Selections, such as menu items or buttons, are listed in *italics*.

Literal code samples are in the `Courier New` typeface with variables in italic: `variable_name`.

The word *click* refers to a single click of the left mouse button. *Right-click* refers to a single click of the right mouse button.

CHAPTER

1

Understanding the Basics

This chapter provides a short overview of the basic parts of a TQL query and the components and optional statements in a query string and includes:

- *TQL Format* on page 4
- *Conjunctions* on page 5
- *Using Property and Category Names* on page 5
- *Using Values* on page 5
- *Using Operators* on page 6

Introducing TQL

The Troux Query Language (TQL) supports two types of queries:

- component queries that return only components
- relationship queries that return only relationships

Queries are built with clauses and conjunctions.

In views, a component query is used to return a set of components. A relationship query may also be used to exclude specific relationships from the view. If no relationship query is specified, then all relationships of the returned components are accessible in the view.

TQL Format

When creating a query, you can use white space and new lines to format the query because the spaces are ignored. Comments are indicated by two dashes (--). The text after the dashes to the end of a line is not evaluated.

To escape quotes and double-quotes in strings:

- Strings surrounded by single quotes use two consecutive single quotes in the string to represent a single quote. For example: 'Select the Computers''s Operating System' will read as Select the Computer's Operating System.
- Strings surrounded by double quotes use two consecutive double-quotes in the string to represent a single double-quote. For example "The Terminal Says, ""Please Enter Your User ID.""" will read as The Terminal Says, "Please Enter Your User ID."

There is no case-sensitivity in TQL. This applies to TQL commands, parameter names, values used with component names, property names and property values.

TQL clauses for component and relationship core attributes use operators and values to specify the criteria for the results.

Queries often use component and relationship clauses together to return sets of components or relationships.

Conjunctions

Conjunctions combine clauses or change the order in which they are evaluated. The following table defines valid TQL conjunctions.

Conjunction	Definition
`clause1 and clause2`	Use **and** to return only those objects that are returned by both *clause1* and *clause2*.
`clause1 or clause2`	Use **or** to return objects that are returned either by *clause1* or by *clause2*.
`not clause1`	Use **not** to return only those objects that are not returned by *clause1*.
`(clause1 or clause2) and clause3`	Use parenthesis to change the order of evaluation. The standard order of evaluation is not, and, then or. For clarification, it is best to always use parenthesis even if the query follows the standard order of evaluation. In this example, the or phrase in the parenthesis would be evaluated before the and phrase.

Using Property and Category Names

Component and relationship properties may be organized into categories and subcategories. Properties are identified by their category and property name using dot-notation. For example, if a component type has a property named "USDollars" in subcategory "Currency" under category "Cost", then the full name of the property would be "Cost.Currency.USDollars".

Note that if a property or category has a dot in its name, those dots must be escaped. For example a property named "first.last" in a category "full.name" would be written "full\.name.first\.last".

Using Values

A value is an expression used in comparisons with component and relationship core attributes. TQL supports the following kinds of values:

- string literals in quotes (`"abc", "17"`)
- numeric literals (`12.568, -15.8, 0`)
- boolean literals (`true / false`)
- datetime literals (`01-01-2004, 12-15-1972`)
- parameters (e.g. `parameter("MachineName")`)
- USERNAME macro (username for current user)

For values, use single or double quotes. See Parameters on page 15 for more information on parameter values.

The USERNAME macro is a variable that evaluates to the user ID specified for user authentication. For example, if Bob Smith logs in as "bsmith" the variable would evaluate to "bsmith". If using the out-of-the-box configuration for user authentication, this is the Tomcat username ID. If using an external authentication tool such as Windows NT, the USERNAME returned is the Windows NT user ID. For example, the clause `component.property("owner") = USERNAME` would return all components where the value of the property named *owner* is equal to the username of the currently logged in user.

Using Operators

Operators are used in clauses to compare values to component and relationship core attributes. The following table lists valid TQL operators.

Operator	Definition
=	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
`contains`	Contains the value as a substring (not available for boolean, datetime, or numeric data)

Operators use the following orderings by default:

- case-insensitive, alphabetical ordering for string literals
- numeric ordering for numeric literals
- numeric ordering for boolean literals (where true=1, and false 0)

Operators can be explicitly included in a clause or a user can select one with an operator picker. For more information see *Working With Parameter Selectors* on page 22.

CHAPTER

2

Working with Components

This chapter covers the TQL clauses for working with component attributes and includes:

- *Working With Component Core Attributes* on page 8
- *Working With Component Hierarchies* on page 10
- *Working With Component Sets* on page 11
- *Working With Relationship in Component Queries* on page 12
- *Working With Component States* on page 13

Understanding Component Queries

Component queries are used to return all components in the system that match a specified component attribute such as a component name, a relationship type, or a range of component attributes defined by values and operators.

Working With Component Core Attributes

Component core attributes include: id, name, description, type, property, and the presence of events. The following clauses compare these core attributes with values and return the components that match the specified criteria.

component.id

Returns all components whose id's match the specified operator and value.

All components have id's that are used as internal keys and are not accessible through the Troux Blueprinting System user interface. However, if accessing the Troux Blueprinting System through the SOAP APIs, you can access these id's and use them in TQL with the following operators:

- `component.id = value`
- `component.id != value`
- `component.id contains value`
- `component.id > value`
- `component.id < value`
- `component.id >= value`
- `component.id <= value`

component.name

Returns all components whose names match the specified operator and value:

- `component.name = value`
- `component.name != value`
- `component.name contains value`
- `component.name > value`
- `component.name < value`
- `component.name >= value`
- `component.name <= value`

component.type

Returns all components whose type matches the specified operator and value. If the type specified has subtypes, all components of that type or of its subtypes are returned.

For example, consider the following set of component types:

- Database
  - SQL
    - 7.0
    - 2K
  - Oracle
    - 7.3.4
    - 8i

With these component types, the clause component.type="Database" returns components of all the types shown; the clause component.type="Oracle" returns components of subtype Oracle, subtype 7.3.4, and subtype 8i; and the clause component.type="8i" returns only components of the subtype 8i. TQL usage for `component.type` includes the following operators and values:

- `component.type = value`
- `component.type != value`
- `component.type contains value`

component.exactType

Returns all components that are of an exact component type (does not include subtypes). TQL usage for usage for `component.exactType` includes the following operators and values:

- `component.exactType = value`
- `component.exactType != value`
- `component.exactType contains value`

component.property

Returns all components that have a property operator and value. A parameter can be used for the property name.

- `component.property("property_name",datatype) = value`
- `component.property("property_name",datatype) != value`
- `component.property("property_name",datatype) contains value`
- `component.property("property_name",datatype) > value`
- `component.property("property_name",datatype) < value`
- `component.property("property_name",datatype) >= value`
- `component.property("property_name",datatype) <= value`

A property can have one of the following data types:

- string - a quoted text string
- numeric - a number value (can have decimal)
- boolean - either true or false
- link - link with a fully-qualified URL (such as http://www.troux.com)
- datetime - a date/time stamp
- Custom property types - lists of defined values. Custom property lists are named and must be called out specifically by name in the format: `propertyType (property_type_name)`.

Properties are arranged in categories and subcategories, and are identified according to their category and property name in dot-notation. For example, if a component type has a property named "USDollars" in subcategory "Currency" under category "Cost", then the full name of the property would be "Cost.Currency.USDollars".

Note that if a property or category has a dot in its name, those dots must be escaped. For example a property named "first.last" in a category "full.name" would be written "full\.name.first\.last".

component.hasEvent

Returns all components that have events with the specified severity or worse (from info through catastrophic).

For example, if you specify "warning" for the severity, components with warning events, failure events, and catastrophic events are returned:

- `component.hasEvent(info)`
- `component.hasEvent(success)`
- `component.hasEvent(warning)`
- `component.hasEvent(failure)`
- `component.hasEvent(catastrophic)`

For more information on events, see the *Troux 4 Blueprint Manager User Guide.*

Working With Component Hierarchies

Components are organized within hierarchies of parent and child levels. The following clauses evaluate components' hierarchical placement and return the components that match the specified criteria.

component.topLevel

Returns all components that have no parent components:

- `component.topLevel`

component.hasAncestor

If component query returns any components, all descendants of those components (at any depth) are returned:

- `component.hasAncestor(`*`component_query`*`)`

component.hasDescendant

If component query returns any components, all ancestors of those components are returned.

- `component.hasDescendant(`*`component_query`*`)`

component.hasParent

If component query returns any components, all of the immediate children of those components are returned.

- `component.hasParent(`*`component_query`*`)`

Working With Component Sets

The following clauses define and return sets of components. Once a set is defined according to a component query and named it can be referenced by that name in other TQL statements. The following clauses compare these sets and return groups of components that match the specified criteria.

define.component.set

When building a complex query, sometimes you may need to repeat the same sub-query several times. Rather than repeating the sub-query, it is easier and more efficient to perform the sub-query once and name it as a set. Then, you can reference the set by name multiple times in your query. You can define any number of sets, but all set definitions must occur at the beginning of a query.

In the component query returns any components, they are defined as a named set:

- `define.component.set("`*`set_name`*`", `*`component_query`*`)`

component.set

Once a component set is defined, this clause returns all components in the named set:

- `component.set("`*`set_name`*`")`

Working With Relationship in Component Queries

Relationships describe the interdependencies that components have with one another. Every relationship is between two components and a component may have multiple relationships each to one other components. A relationship defines which of the two components is dependent on the other: first depends on the second, the first is required for the second, neither is dependent on the other, or both are dependent on the other.

component.hasRelationship

If relationship query returns any relationships, then `component.hasRelationship` returns all components attached to those relationships:

- `component.hasRelationship(`*`relationship_query`*`)`

component.relatedTo

If the component query returns any components, then `component.relatedTo` returns those components as well as all components related to them:

- `component.relatedTo(`*`component_query, numberOfLevels`*`)`

`numberOfLevels` is optional; the default number of levels is 1. If `numberOfLevels` is 1, the components' immediate relationships are returned; if `numberOfLevels` is 2, the relationships two levels out are returned (all components related to immediate related items), and so on. Use a very large number to return all related components in the blueprint.

component.dependsOn

If the component query returns any components, then `component.dependsOn` returns those components as well as the components that depend on them

- `component.dependsOn(`*`component_query, numberOfLevels`*`)`

If `numberOfLevels` is 1, the components' immediately related components are returned; if `numberOfLevels` is 2, the components related two-levels out are returned; (all components related to immediate related items), and so on. Use a very large number to return all related components in the blueprint.

component.requiredFor

If the component query returns any components, then `component.requiredFor` returns those components as well as the components on which they depend:

- `component.requiredFor(`*`component_query, numberOfLevels`*`)`

If `numberOfLevels` is 1, the immediately related components are returned; if `numberOfLevels` is 2, the components related two-levels out are returned; etc.

Working With Component States

The following clauses return components of specific states. Components can be in abstract, disabled or invalid states.

component.abstractType

Typically, component instances of abstract component types will not be used in the system. However, the following clauses will return components that are of any abstract type.

This query returns all components in the system that are of an abstract type:

- `component.abstractType`

component.disabledType

Typically, component instances of disabled types will not be used in the system. However, the following clause will return components of any disabled types.

This query returns all components in the system that are of a disabled type:

- `component.disabledType`

component.hasMissingRequiredProperty

This query returns components that are missing one or more required property values.

- `component.hasMissingRequiredProperty`

component.hasInvalidProperty

This query returns components that have one or more invalid property values:

- `component.hasInvalidProperty`

CHAPTER

3

Working With Relationships

This chapter covers the TQL clauses for working with relationship attributes and includes:

- *Working With Relationship Core Attributes* on page 16
- *Working With Relationships' Components* on page 18
- *Working With Relationship Sets* on page 18
- *Working With Relationship States* on page 19

Understanding Relationship Queries

Relationship queries return all relationships in the system that match a specified relationship attribute such as a relationship name, an associated component type, or a range of component attributes defined by values and operators.

Working With Relationship Core Attributes

Relationship core attributes include: id, description, type, property, and the presence of events. The following clauses compare these core attributes with values and return the relationships that match the specified criteria.

relationship.id

Returns all relationships whose id's match the specified operator and value.

All relationships have id's that are used as internal keys and are not accessible through the Troux Blueprinting System user interface. However, if you are accessing the Troux Blueprinting System through the SOAP APIs, you have access to these id's and can use them in TQL:

- `relationship.id = value`
- `relationship.id != value`
- `relationship.id contains value`
- `relationship.id > value`
- `relationship.id < value`
- `relationship.id >= value`
- `relationship.id <= value`

relationship.type

Returns all relationships whose type matches the specified operator and value. If the type specified has subtypes, all relationships of that type or of its subtypes are returned:

- `relationship.type = value`
- `relationship.type != value`
- `relationship.type contains value`

relationship.exactType

Returns all relationships that are of an exact relationship type (does not include subtypes):

- `relationship.exactType = value`
- `relationship.exactType != value`
- `relationship.exactType contains value`

relationship.property

Returns all relationships that have a specified property that matches the specified operator and value. A parameter can be used for the property name.

- `relationship.property("property_name", datatype) = value`
- `relationship.property("property_name", datatype) != value`
- `relationship.property("property_name", datatype) contains value`
- `relationship.property("property_name", datatype) < value`
- `relationship.property("property_name", datatype) > value`
- `relationship.property("property_name", datatype) <= value`
- `relationship.property("property_name", datatype) >= value`

A property can have one of the following data types:

- string - a quoted text string
- numeric - a number value (can have decimal)
- boolean - either true or false
- link - link with a fully-qualified URL (such as http://www.troux.com)
- datetime - a date/time stamp
- Custom property types - lists of defined values. Custom property lists are named and must be called out specifically by name in the format: `propertyType (property_type_name)`.

Properties are arranged in categories and subcategories, and are identified according to their category and property name in dot-notation. For example, if a component type has a property named "USDollars" in subcategory "Currency" under category "Cost", then the full name of the property would be "Cost.Currency.USDollars".

Note that if a property or category has a dot in its name, those dots must be escaped. For example a property named "first.last" in a category "full.name" would be written "full\.name.first\.last".

relationship.hasEvent

Returns all relationships that have events with the specified severity or worse (from info to catastrophic).

For example, if you specify "warning" for the severity, relationships with warning events, failure events, and catastrophic events are returned:

- `relationship.hasEvent(info)`
- `relationship.hasEvent(success)`
- `relationship.hasEvent(warning)`
- `relationship.hasEvent(failure)`
- `relationship.hasEvent(catastrophic)`

Working With Relationships' Components

Component interactions are defined by relationships. The following clauses evaluate and return the relationships that match the specified criteria for a specified set of components.

relationship.hasComponent

If component query returns any components, relationship.hasComponent returns all relationships attached to those components:

- `relationship.hasComponent(`*`component_query`*`)`

relationship.traceRelationshipsFrom

If the component query returns any components, `relationship.traceRelationshipsFrom` traces out and returns the specified types of relationships from the components for a given direction and the specified number of levels.

If "`all`" is specified for the direction, all relationships are returned. If "`dependsOn`" is specified, only relationships that specify dependencies on other components are returned. If "`requiredFor`" is specified, only relationships that specify dependencies from other components are returned (that is, the opposite of `dependsOn`).

If `numberOfLevels` is 1, the components' immediate relationships are returned; if `numberOfLevels` is 2, the relationships two-levels out are returned; etc. To return all relationships at any level use a very large number.

- `relationship.traceRelationshipsFrom(`*`component_query`*`, all | dependsOn | requiredFor, `*`numberOfLevels`*`)`

Working With Relationship Sets

The following clauses define and return sets of relationships.

define.relationship.set

When building a complex query, sometimes you need to repeat the same sub-query several times. Rather than repeating the sub-query, it is easier and more efficient to perform the sub-query once and name it as a set. Then, you can reference the set by name multiple times in your query. You can define any number of sets, but all set definitions must occur at the beginning of a query:

- `define.relationship.set("`*`set_name`*`", `*`relationship_query`*`)`

relationship.set

Once the relationship set is defined, this clause returns all relationships in the named set. See `define.relationship.set` for more information:

- `relationship.set("`*`set_name`*`")`

Working With Relationship States

The following clauses return relationships of a certain state.

relationship.disabledType

A disabled relationship type may be a relationship that depends on or uses a disabled component type. For example, if a new software package is planned for production, relationships for computer that will run this new software can be created in the system as disabled types until the software package is ready for use.

This query returns all relationships in the system that are of a disabled type:

- `relationship.disabledType`

relationship.hasMissingRequiredProperty

This query returns components that are missing one or more required property values.

- `relationship.hasMissingRequiredProperty`

relationship.hasInvalidProperty

This query returns components that have one or more invalid property values:

- `relationship.hasInvalidProperty`

CHAPTER

4

Using Parameters and Parameter Selectors

This chapter covers TQL statements used to create and return parameters associated with components and relationships and includes:

- *Working With Parameters* on page 22
- *Working With Parameter Selectors* on page 22
- *Working with Data Type-Dependent Selectors* on page 26

Working With Parameters

TQL allows you to define parameters and use parameter values. Using parameters means that you can write a single query that can be selectively applied to many objects.

For example, suppose your blueprint contains 100 applications. For each application, you want to have a view that shows you the application and everything related to it. You could either write 100 queries or you could write a single query that takes the name of the application as a parameter.

Parameters can also depend on other parameters. If a parameter is defined using other parameters, then they are grayed in the user interface until a user specifies the other values required. For example, If a query includes a parameter called *location*, and *location* is defined by *building*, *floor*, and *room*, then the user can be prompted to enter or select the building, then the floor, and then the room to provide the information needed to define the parameter *location*.

define.parameter

Defines a parameter with the specified name. The Troux Blueprinting System prompts the user to enter values in a text box or an from optional parameter selector:

- `define.parameter("parameter_name", "prompt" [, parameter selector])`

For more information, see Working With Parameter Selectors on page 22.

parameter

Once a parameter is defined, this clause retrieves the parameter's value. Parameter values can be used in place of literal values for query types and other functions:

- `parameter("parameter_name")`

The following is a simple example of using a parameter:

```
define.parameter("compname", "Enter a component name")
component.relatedTo(component.name = parameter("compname"))
```

With this query, the user is prompted to "Enter a component name." Once the user enters the name and clicks Submit, the query uses the entered name as a value in the component query.

Working With Parameter Selectors

Using parameter selectors within parameter definitions allows you to present the user with a list of options to choose from rather than forcing the user to know and enter a valid value into a text box.

selectOperator

This parameter can be used any place in TQL where a comparison operator is allowed, presenting a list of possible operators to the user of the query.

- `selectOperator()`

For example, when a user runs a query with the following:

```
define.parameter("operator","Select an operator", selectOperator())
component.name parameter("operator") "Troux"
```

The user is presented a drop-down list from which to select one of the following operators. The selected operator is used in the component query:

- equal to
- not equal to
- less than
- greater than
- less than or equal to
- greater than or equal to

Note *contains* is not included in selectable operators.

selectDate

Displays a calendar-formatted date selection tool:

- `selectDate()`

The following example opens a `selectDate` parameter selection dialog that prompts the user to "Select a Date" and then uses that value in a component query.

```
define.parameter("SomeDate", "Select a Date", selectDate())
component.property("Dates.Purchase_Date") = parameter("SomeDate")
```

Note Parameter selectors for existing and possible property values present the user with a selector as well as an icon for changing that selector.

Selecting existing values presents existing property value by default:

Selecting possible values presents an input box by default:

If the number of values returned is greater than the number set for the *maxParameterOptions* parameter found in *baseConfig.xml* then the user is presented with a selector appropriate to the data type for the possible range of values rather than a list of all existing values. For more information on the setting, see the *Configuration Reference*.

selectExistingComponentPropertyValue

Displays a list of all values of the named property for all components, or all values of the named property for components returned by the optional component query.

- `selectExistingComponentPropertyValue("property_name", datatype, [component_query])`

A selection list displays that has all existing values for the named property, as shown in this example:

```
define.parameter("PickState", "Select State",
selectExistingComponentPropertyValue("State", string, component.type="Location"))
component.type="Location" and
component.property("State", string) = parameter("PickState")
```

selectPossibleComponentPropertyValue

Displays the appropriate input format for entering a value for the property, based on the selected property's data type.

- `selectPossibleComponentPropertyValue("property_name", datatype, [component_query])`

In example:

```
define.parameter("PickState", "Select State",
selectPossibleComponentPropertyValue("State", string, component.type="Location"))
component.type="Location" and
component.property("State", string) = parameter("PickState")
```

selectExistingRelationshipPropertyValue

Displays a list of all values of the named property for all relationships, or all values of the named property for relationships returned by the optional relationship query.

- `selectExistingComponentPropertyValue("property_name", datatype, [relationship_query])`

In example:

```
define.parameter("PickDate", "Select a date",
selectExistingRelationshipPropertyValue("DateEst", datetime, relationship.type="RunsOn"))
relationship.property("DateEst", datetime) = parameter("PickDate")
```

selectPossibleRelationshipPropertyValue

Displays the appropriate input format for entering a value for the property, based on the selected property's data type.

- `selectPossibleRelationshipPropertyValue("property_name", datatype, [relationship_query])`

In example:

```
define.parameter("MonthYear", "Pick a Month and Year",
selectPossibleRelationshipPropertyValue("Date Established", datetime,
relationship.type="RunsOn"))
relationship.type = "RunsOn" and
relatinships.property("Date Established", datetime) = parameter("MonthYear")
```

selectComponentType

Displays the components that are of a specific type (including its subtypes) and optionally, of a specific state (abstract, disabled, notAbstract, notDisabled, or abstractOrDisabled.

- `selectComponentType("component_type" [, abstract | disabled | notAbstract | notDisabled | abstractOrDisabled ])`

Presents a list of component type and its subtypes. If "abstract" is specified in selectComponentType, only abstract types are shown in the type selector. If "disabled" is specified, only disabled types are shown and so on.

```
define.parameter("AbsCompType", "Select an abstract component Type",
selectComponentType("Generic Component", abstract)
component.type = parameter("AbsComptype")
```

selectRelationshipType

Displays the relationships that are of a specific type and optionally, of a specific state (disabled or notDisabled).

- `selectRelationshipType("relationship_type" [, disabled | notDisabled ])`

Presents a list of relationship type and its subtypes. If "disabled" is specified, only disabled types are shown.

```
define.parameter("DisRelated", "Select a disabled relationship type",
selectRelationshipType("Generic Relationship", disabled))
relationship.type = parameter("DisRelated")
```

enumeration

Displays a list of selections, from which the user can select one or more items. An enumeration is named during creation, and the *enumeration Name* must be used in queries. The parameter is assigned the value(s) of the selection(s).

- `enumeration(("selection1","value1") [, ("selection2","value2"), ...] )`

The following query allows the user to select an item from a text-based list to set a value:

```
define.parameter("pStatus", "Application Status",
  enumeration(("Development","1"),("Production","2"),("Retired","3")))
```

The clause `define.parameter` is used to define the parameter `pStatus`. The parameter selector *`enumeration`* populates a list of selections. One or more of these names is selected by the user and becomes the value(s) of the parameter *`pStatus`*. The parameter can then be used in a parameter clause to retrieve the parameter's value.

Working with Data Type-Dependent Selectors

For some queries the user must pick a property definition (such as "width" or "region") as a parameter. In some cases, the property that the user selects may require a datatype to resolve query ambiguity. Because of this, the results of the parameter selectors are special types of parameter values. The examples for the following selectors include statements that illustrate this requirement.

selectComponentPropertyByType

Displays the components with properties of a specified type selected.

- `selectComponentPropertyByType(parameter("type"))`

The following query restricts the list of existing values to those on components of the chosen types and allows the user to select a property. In cases where multiple data types have the same property name, it uses the appropriate type:

```
define.parameter("Type", selectComponentType("Generic Component")
define.parameter("Property", selectComponentPropertyByType(parameter("Type")))
define.parameters("Value",selectExistingComponentPropertyValue(parameter("Proper
ty"),component.type = parameter ("Type")))
```

This query prompts the user to choose a type, property and then a value for that property and returns the components that match the criteria:

```
component.type = parameter ("Type") and
component.property(parameter("Property")) = parameter("Value")
```

selectRelationshipPropertyByType

Displays the relationships with properties of a specified type that is selected by the user

- `selectRelationshipPropertyByType(parameter("type"))`

The following query restricts the list of existing values to those on relationships of the chosen types and allows the user to select a property. In cases where multiple data types have the same property name, it uses the appropriate type:

```
define.parameter("type", selectRelationshipType("Generic Relationship")
define.parameter("property",
selectRelationshipPropertyByType(parameter("type")))
define.parameters("value",selectExistingRelationshipPropertyValue(parameter("pro
perty"),relationship.type = parameter ("type")))
```

This query prompts the user to choose a type, property and then a value for that property and returns the relationships that match the criteria:

```
relationship.type = parameter ("type") and
relationship.property(parameter("property")) = parameter("value")
```

The parameter values defined by these three parameter selectors require a particular context to be useful. This can only be used when a property is *always* expected, and not an other entity such as a component or a relationship:

```
define.parameter("prop",selectComponentPropertyByType("Computer"))
component.type  = parameter("prop")
```

This example would not work because the parameter *prop* is defined as a property in the first statement not a component.